Date: 03 - 01 - 2026
Assignment No.: 1

Name: Varghese K James
Reg No.: 3122245001192

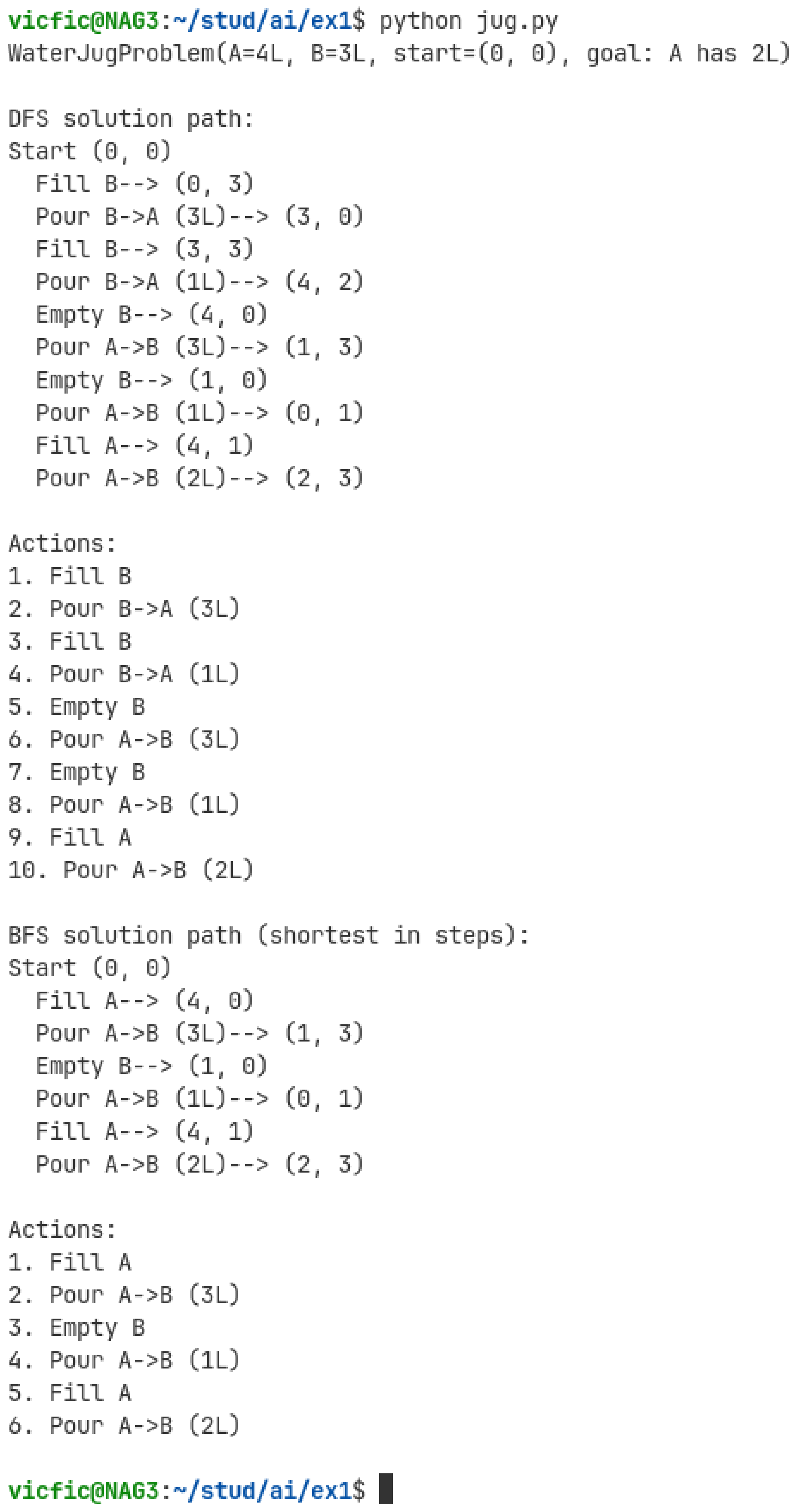
Excercise 1a - Uninformed Search Strategy - Depth First Search

Aim: Implement Depth First Search (DFS) for the Water Jug problem.

Source Code

```
from collections import deque

class SearchProblem:
    def start_state(self):
        pass

    def is_goal(self, state):
        pass

    def neighbors(self, state):
        pass

class Edge:
    def __init__(self, from_state, to_state, action, cost=1):
        self.from_state = from_state
        self.to_state = to_state
        self.action = action
        self.cost = cost

    def __repr__(self):
        return f"Edge({self.from_state} -> {self.to_state}, 
action='{self.action}', cost={self.cost})"

class WaterJugProblem(SearchProblem):
    def __init__(self, cap_a=4, cap_b=3, start=(0, 0), 
goal_amount_in_a=2):
        self.cap_a = cap_a
        self.cap_b = cap_b
        self.start = start
        self.goal_amount_in_a = goal_amount_in_a

    def __repr__(self):
        return (
            f"WaterJugProblem(A={self.cap_a}L, B={self.cap_b}L, "
            f"start={self.start}, goal: A has
{self.goal_amount_in_a}L)"
        )

    def start_state(self):
        return self.start

    def is_goal(self, state):
        x, y = state
        return x == self.goal_amount_in_a

    def neighbors(self, state):
        x, y = state
        edges = []
        seen = set()

        def add(to_state, action):
            if to_state != state and to_state not in seen:
                seen.add(to_state)
                edges.append(Edge(state, to_state, action, 1))

        add((self.cap_a, y), "Fill A")
        add((x, self.cap_b), "Fill B")
        add((0, y), "Empty A")
        add((x, 0), "Empty B")

        pour = min(x, self.cap_b - y)
        add((x - pour, y + pour), f"Pour A->B ({pour}L)")

        pour = min(y, self.cap_a - x)
        add((x + pour, y - pour), f"Pour B->A ({pour}L)")

        return edges


class Path:
    def __init__(self, start_state, prev=None, edge=None):
        self.start_state = start_state
        self.prev = prev
        self.edge = edge
    def end(self):
        if self.edge is None:
            return self.start_state
        return self.edge.to_state

    def __repr__(self):
        seq = self.states_and_edges()
        lines = [f"Start {seq[0][0]}"]
        for st, ed in seq[1:]:
            lines.append(f"  {ed.action}--> {st}")
        return "\n".join(lines)

    def states_and_edges(self):
        parts = []
        cur = self
        while True:
            if cur.edge is None:
                parts.append((cur.start_state, None))
                break
            parts.append((cur.edge.to_state, cur.edge))
            cur = cur.prev
        parts.reverse()
        return parts

    def actions(self):
        seq = self.states_and_edges()
        return [ed.action for _, ed in seq[1:]]


def dfs(problem):
    start = problem.start_state()
    frontier = [Path(start)]
    explored = set()

    while frontier:
        path = frontier.pop()
        s = path.end()

        if problem.is_goal(s):
            return path

        if s in explored:
            continue
        explored.add(s)

        for e in problem.neighbors(s):
            if e.to_state not in explored:
                frontier.append(Path(start, prev=path, edge=e))
    return None


def bfs(problem):
    start = problem.start_state()
    frontier = deque([Path(start)])
    explored = set([start])

    while frontier:
        path = frontier.popleft()
        s = path.end()

        if problem.is_goal(s):
            return path

        for e in problem.neighbors(s):
            if e.to_state not in explored:
                explored.add(e.to_state)
                frontier.append(Path(start, prev=path, edge=e))
    return None



def print_solution(path, title):
    print(title)
    if path is None:
        print("No solution found.")
        return
    print(path)
    print("\nActions:")
    for i, a in enumerate(path.actions(), 1):
        print(f"{i}. {a}")
    print()


if __name__ == "__main__":
    problem = WaterJugProblem(cap_a=4, cap_b=3, start=(0, 0),
goal_amount_in_a=2)
    print(problem)
    print()

    dfs_path = dfs(problem)
    print_solution(dfs_path, "DFS solution path:")

    bfs_path = bfs(problem)
    print_solution(bfs_path, "BFS solution path (shortest in
steps):")
```

Input/Output



```
vicfic@NAG3:~/stud/ai/ex1$ python jug.py
WaterJugProblem(A=4L, B=3L, start=(0, 0), goal: A has 2L)

DFS solution path:
Start (0, 0)
  Fill B--> (0, 3)
  Pour B->A (3L)--> (3, 0)
  Fill B--> (3, 3)
  Pour B->A (1L)--> (4, 2)
  Empty B--> (4, 0)
  Pour A->B (3L)--> (1, 3)
  Empty B--> (1, 0)
  Pour A->B (1L)--> (0, 1)
  Fill A--> (4, 1)
  Pour A->B (2L)--> (2, 3)

Actions:
1. Fill B
2. Pour B->A (3L)
3. Fill B
4. Pour B->A (1L)
5. Empty B
6. Pour A->B (3L)
7. Empty B
8. Pour A->B (1L)
9. Fill A
10. Pour A->B (2L)

BFS solution path (shortest in steps):
Start (0, 0)
  Fill A--> (4, 0)
  Pour A->B (3L)--> (1, 3)
  Empty B--> (1, 0)
  Pour A->B (1L)--> (0, 1)
  Fill A--> (4, 1)
  Pour A->B (2L)--> (2, 3)

Actions:
1. Fill A
2. Pour A->B (3L)
3. Empty B
4. Pour A->B (1L)
5. Fill A
6. Pour A->B (2L)

vicfic@NAG3:~/stud/ai/ex1$ 
```

Learning Outcomes

- I am able to implement uninformed search strategy using DFS and BFS.
- I am able to solve the Water Jug problem using search strategy.